

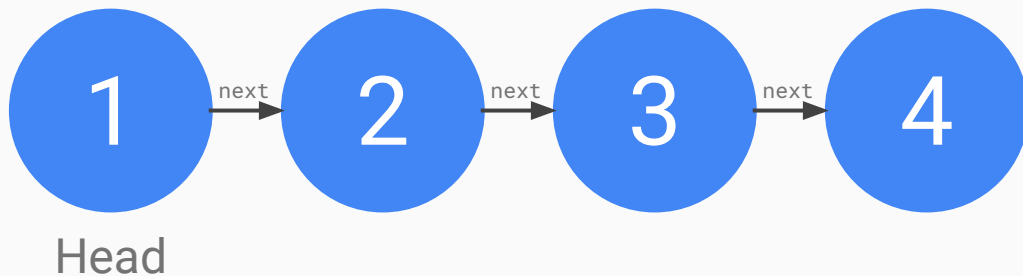
Linked lists and hashing

Code: <https://github.com/mesmere/sasha-tutorial/tree/main/2024-02-25>

Linked lists

Linked lists - Introduction

- Efficient at (head) insertions and at deletions.
- Conventionally *never* used in JavaScript — but we'll do it anyway.



```
1 class Node {
2   data;
3   next = undefined;
4
5   constructor(data) {
6     this.data = data;
7   }
8 }
9
10 const head = new Node(1);
11 head.next = new Node(2);
12 head.next.next = new Node(3);
13 head.next.next.next = new Node(4);
14
15 console.log(JSON.stringify(head, undefined, "  "));
```

```
{
  "data": 1,
  "next": {
    "data": 2,
    "next": {
      "data": 3,
      "next": {
        "data": 4
      }
    }
  }
}
```

Linked lists - Traversing

Get the k th element from a linked list:

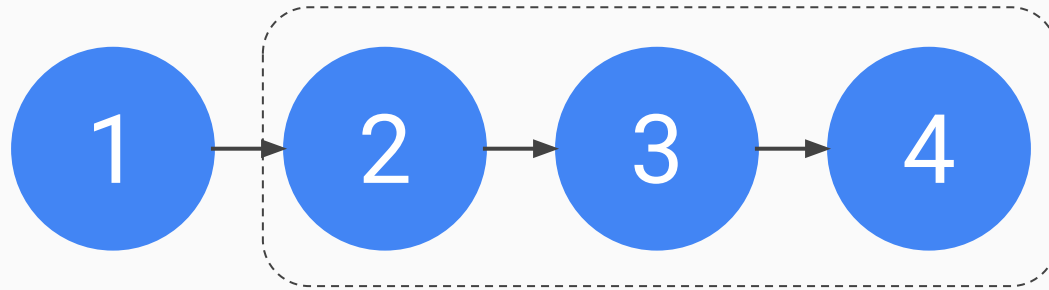
1. Start at the head.
2. Follow the next reference k times.

NB: `cur` is a node, not a numerical index.

```
19 get(k) {  
20   if (this.head === undefined) {  
21     throw "Out of range.";  
22   }  
23  
24   let cur = this.head;  
25   while (--k >= 0) {  
26     if (cur.next === undefined) {  
27       throw "Out of range.";  
28     }  
29     cur = cur.next;  
30   }  
31   return cur.data;  
32 }
```

Linked lists - Traversing (recursive)

Wait a second, lists have a recursive structure...



This part is itself a linked list with "2" at its head!

Linked lists - Traversing (recursive)

Get the k th element from a linked list recursively:

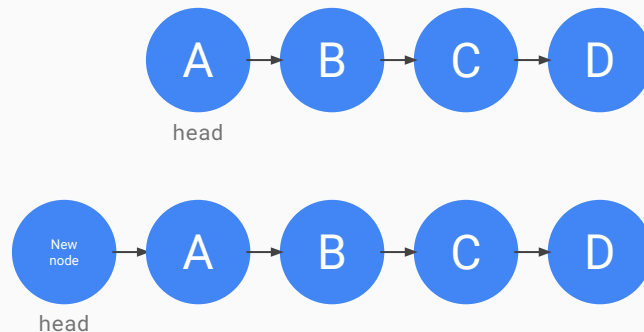
1. Start at the head.
2. Get the $(k-1)$ th element from the “list” which is located at `head.next`.

```
24 getRecursive(k) {  
25   function helper(node, k) {  
26     if (node === undefined) {  
27       throw "Out of range."  
28     }  
29     if (k === 0) {  
30       return node.data;  
31     }  
32     return helper(node.next, k-1);  
33   }  
34  
35   return helper(this.head, k);  
36 }
```

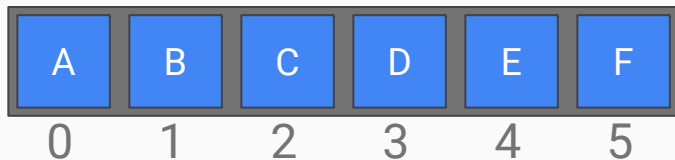
Linked lists (inserting)

Operations at the head of the list are $O(1)$:

```
38 insertAtHead(data) {  
39   const oldHead = this.head;  
40   this.head = new Node(data);  
41   this.head.next = oldHead;  
42 }
```



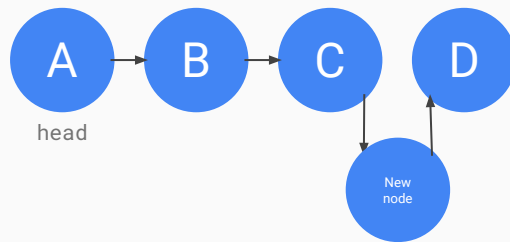
Compare this “ $O(n)$ ” insertion at the head of an array (shift n elements):



Linked lists (inserting)

To insert at the k th position in the list, we'll have to traverse to get there:

```
44 insert(k, data) {  
45   // Special case to update head  
46   if (k === 0) {  
47     this.insertAtHead(data);  
48     return;  
49   }  
50  
51   if (this.head === undefined) {  
52     throw "Out of range.";  
53   }  
54  
55   let cur = this.head;  
56   while (--k >= 1) { // Stop short!  
57     if (cur.next === undefined) {  
58       throw "Out of range.";  
59     }  
60     cur = cur.next;  
61   }  
62  
63   const oldNext = cur.next;  
64   cur.next = new Node(data);  
65   cur.next.next = oldNext;  
66 }
```



We stop with `cur` at the $(k-1)$ th node...

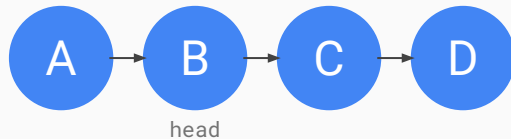
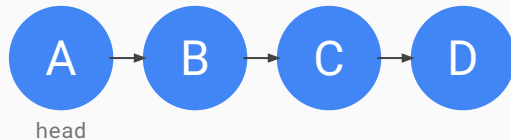
...and set `cur . next` to our new node.

Linked lists (removing)

What about removing elements?

Again, operations at the head are easy:

```
68 removeAtHead() {  
69   if (this.head === undefined) {  
70     throw "Out of range.";  
71   }  
72   this.head = this.head.next;  
73 }
```



“A” is still pointing to “B”, but there’s no way to reach “A” now. The garbage collector will free up unreachable parts of the object graph eventually...

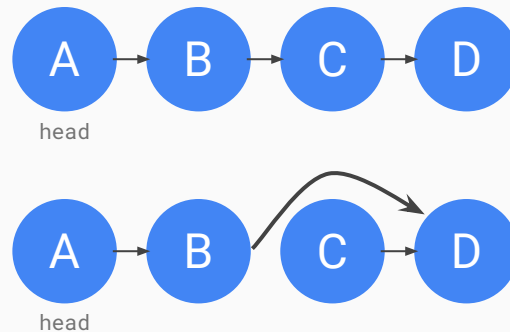
Linked lists (removing)

To remove the k th item in the list, we'll have to traverse to get there:

```
75 remove(k) {  
76   // Special case to update head  
77   if (k === 0) {  
78     this.removeAtHead();  
79     return;  
80   }  
81  
82   if (this.head === undefined) {  
83     throw "Out of range.";  
84   }  
85  
86   let cur = this.head;  
87   while (--k >= 1) { // Stop short!  
88     if (cur.next === undefined) {  
89       throw "Out of range.";  
90     }  
91     cur = cur.next;  
92   }  
93  
94   cur.next = cur.next.next;  
95 }
```

Stop with `cur` at the
($k-1$)th node...

...and set `cur.next` to **its** next.



"C" is still pointing to "D",
but, again, who cares?

Linked lists (looping over)

Let's wrap this thing up in a class so that users can just treat it as a list rather than worrying about traversing nodes. So far we have these methods:

- Get the element at the k^{th} position
- Insert a new element at the k^{th} position
- Remove the element at k^{th} position

But if we leave it here, looping over all of the elements of the list will look like:

```
for(let i=0; i<n; i++)  
  console.log(list.get(i));
```

← $O(n^2)$ right? 😬

Linked lists (looping over)

```
107 * [Symbol.iterator]() {  
108   let cur = this.head;  
109   while (cur !== undefined) {  
110     yield cur.data;  
111     cur = cur.next;  
112   }  
113 }
```

```
const list = new LinkedList(["zero", "one", "two"]);  
for (let cur of list) {  
  console.log(cur);  
}
```

- Putting a name in [brackets] like this lets you evaluate an *expression* to use as a function/method or variable name.
- `Symbol.iterator` evaluates to the name of the **special method** that's invoked on `list` when you do `for (const cur of list)`.
- This method should return an object implementing the [iterable protocol](#). Luckily, **generators** implement this protocol for us.
- Beginning a function/method name with an asterisk indicates a generator.
- Inside a generator function we can use the `yield` keyword to return a value. But the next time we invoke the function, execution **resumes with the next statement** as if the function never returned!

Linked lists (reversing)

```
reverse() {  
  if (this.head === undefined) {  
    return; // Nothing to do.  
  }  
  
  let prev = this.head;  
  let cur = this.head.next;  
  while (cur !== undefined) {  
    const next = cur.next;  
    cur.next = prev;  
    prev = cur;  
    cur = next;  
  }  
  this.head.next = undefined;  
  this.head = prev;  
}
```



Start with **prev** = A, **cur** = B.

Set **B.next** = A.

Next up is **prev** = B, **cur** = C.

Set **C.next** = B.

Next up is **prev** = C, **cur** = D.

Set **D.next** = C.

We're done with the loop. A bit of cleanup:

Set **A.next** = **undefined** and

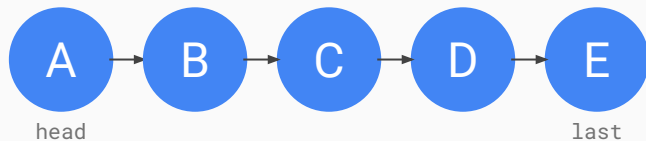
Set **this.head** = E.

Doubly-linked lists

Problem: What if we're frequently doing operations at the end of the list?
Ideally we wouldn't need to traverse all n elements every time.

Solution: In addition to head, keep another reference to the last element.

But how would we delete the last element?

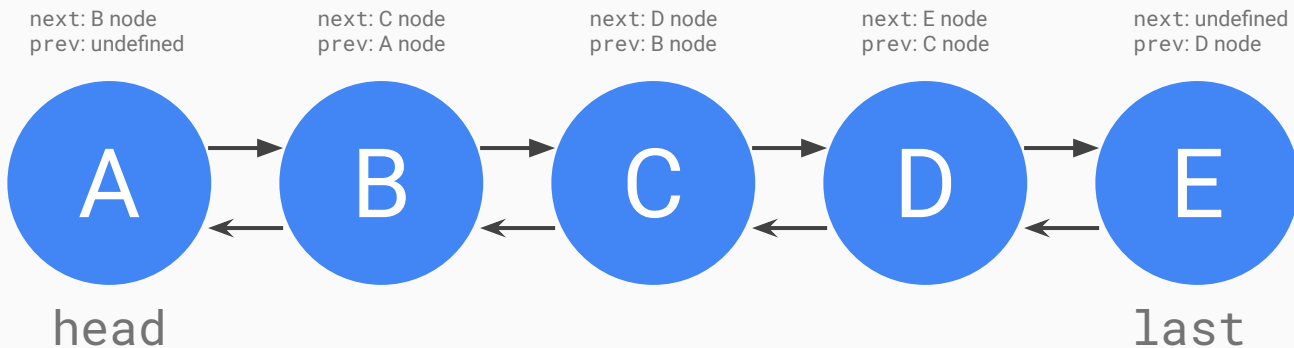


To delete "E" we need to set `D.next = undefined`
...but there's no way to get to "D" without traversing the list.

```
> const list = new LinkedList(['A', 'B', 'C', 'D', 'E']);  
undefined  
> list.last;  
Node { data: 'E', next: undefined }
```

Doubly-linked lists

Simultaneously keep track of next and prev references on each node:



Now to delete node, we just set `node.prev.next = node.next`.

Hashing

Hashing

A **hash function** is a mathematical function which maps arbitrary input onto small numbers (“hashes”).

Example: sha1 is a hash function which takes any data as input and outputs a 20-byte number (conventionally rendered in hexadecimal).

We can feed it text or even a 5MB png file and it will *always* output 20 bytes.

```
~ $ echo "hi there I'm rileyann" | sha1sum
e8793e81bfd2a9122bca404a313175b4e23f28e9 -
~ $ echo "it's very nice to meet you" | sha1sum
030dcebbaf0b3867f136265645b1d56958fa2460 -
~ $ viu -w 30 komeiji-satori.png
```



```
~ $ sha1sum komeiji-satori.png
e8c190de829f878322b887ec764902c3408b16bd komeiji
```

Hashing

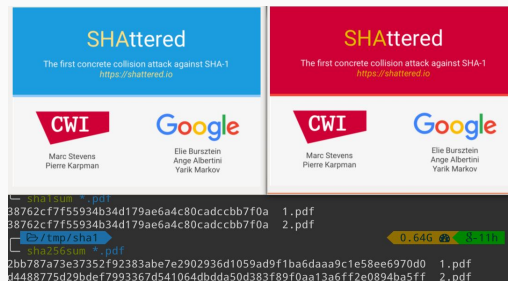
Good hash functions will be:

- **Deterministic.** They always output the same hash for the same input.
- **Sensitive to changes in the input.** A single bit changed in the input will usually change the hash, even if the input is large.
- **Good at distributing inputs *uniformly* across the possible outputs.** It's easy to write hash functions that produce distributions like [this](#) 🤨.
- **Fast.** People put a *lot* of effort into engineering the tradeoff between computational efficiency and the other properties.

Hashing

Sometimes hash functions need to stand up against a malicious attacker. In this kind of scenario these properties are also important:

- **Hard to reverse.** We shouldn't be able to recover the input from the hash.
- **Resistant to collision.** We shouldn't be able to find two inputs that hash to the same value. E.g. a team at Google famously managed to craft two different PDFs with the *same* sha1 hash:



Hashing

In practice, very *poor* hash functions are often used in cases where speed is more important than the other properties.

```
31 static size_t keyhash(char *k)
32 {
33     unsigned char *p = (void *)k;
34     size_t h = 0;
35
36     while (*p)
37         h = 31*h + *p++;
38     return h;
39 }
```

musl libc (*fast*)

```
b32 = (s[11] << 4) | (s[10] >>> 28); b23 = (s[15] << 6) | (s[14] >>> 26);
b33 = (s[10] << 4) | (s[11] >>> 28); b4 = (s[25] << 11) | (s[24] >>> 21);
b14 = (s[20] << 3) | (s[21] >>> 29); b5 = (s[24] << 11) | (s[25] >>> 21);
b15 = (s[21] << 3) | (s[20] >>> 29); b36 = (s[34] << 15) | (s[35] >>> 17);
b46 = (s[31] << 9) | (s[30] >>> 23); b37 = (s[35] << 15) | (s[34] >>> 17);
b47 = (s[30] << 9) | (s[31] >>> 23); b18 = (s[45] << 29) | (s[44] >>> 3);
b28 = (s[40] << 10) | (s[41] >>> 14); b19 = (s[44] << 29) | (s[45] >>> 3);
b29 = (s[41] << 10) | (s[40] >>> 14); b10 = (s[6] << 26) | (s[7] >>> 4);
b20 = (s[2] << 1) | (s[3] >>> 31); b11 = (s[7] << 26) | (s[6] >>> 4);
b21 = (s[3] << 1) | (s[2] >>> 31); b42 = (s[17] << 23) | (s[16] >>> 9);
b2 = (s[13] << 12) | (s[12] >>> 20); b43 = (s[16] << 23) | (s[17] >>> 9);
b3 = (s[12] << 12) | (s[13] >>> 20); b24 = (s[26] << 25) | (s[27] >>> 7);
b34 = (s[22] << 10) | (s[23] >>> 22); b25 = (s[27] << 25) | (s[26] >>> 7);
b35 = (s[23] << 10) | (s[22] >>> 22); b6 = (s[36] << 21) | (s[37] >>> 11);
b16 = (s[33] << 13) | (s[32] >>> 19); b7 = (s[37] << 21) | (s[36] >>> 11);
b17 = (s[32] << 13) | (s[33] >>> 19); b38 = (s[47] << 24) | (s[46] >>> 8);
b48 = (s[42] << 2) | (s[43] >>> 30); b39 = (s[46] << 24) | (s[47] >>> 8);
b49 = (s[43] << 2) | (s[42] >>> 30); b30 = (s[5] << 27) | (s[6] >>> 5);
b40 = (s[5] << 30) | (s[4] >>> 2); b31 = (s[9] << 27) | (s[8] >>> 5);
b41 = (s[4] << 30) | (s[5] >>> 2); b12 = (s[18] << 26) | (s[19] >>> 12);
b22 = (s[14] << 6) | (s[15] >>> 26); b13 = (s[10] << 26) | (s[11] >>> 12);
```

One small piece of the hash
function used by Ethereum
(*secure*)

Application - HMACs

The last time that Alice saw Bob in person she whispered a secret key in his ear: “IMURS4EVER”

One day, Alice wants to DM a message to Bob but it would be nice if:

1. Bob could be sure that Alice is sending a message, i.e. that she's not asleep and it's just some bored fed fucking with him. (*Authenticity*)
2. Bob could be sure that Alice's original message hasn't been maliciously altered in transit. (*Integrity*)

Solution: Along with the message send `sha3(' IMURS4EVER ' +message)`.

Application - HMACs

Bob receives a message and then some hash-looking string in a DM.

Since he already has the shared key IMURS4EVER he can validate the message by *himself* evaluating `sha3(' IMURS4EVER ' +message)`. If the result matches the hash from the DM, then the message is genuinely from Alice!

Why does this work?

- Any attacker intercepting their DMs **cannot reverse a hash function** to recover the input value (' IMURS4EVER ' +message) from the hash to reveal the key.
- Without the key, if an attacker alters the message they **can't provide the corresponding hash value** `sha3(' IMURS4EVER ' +evilMessage)`.

Application - Hash tables

Problem: Store a set of keys and values such that all of the keys are unique — and provide **efficient** lookup of the value that's associated with any given key.

(This is Map from JavaScript, or just regular JS objects.)

Naive solution: Store all of the <key, value> pairs in an array sorted by key and do binary search every time to look up the corresponding value.

Better solution: Allocate a big array and **hash the keys** to determine *exactly* where in the array their corresponding values should be located, with no* searching necessary. 😎

Application - Hash tables (Learn by example!)

Let $\text{hash}(\text{str}) = \text{sum of the ASCII codes of the characters of str}$ (so A=65, B=66, ...).

Our table will be an array of size $n=10$, so we'll be able to hold up to 10 key/value pairs.

Let's try inserting Key: 'MOON', Value: 'cheese'.

First calculate the hash of the key: $\text{hash}('MOON') = 77+79+79+78 = 313$.

We need to map this into our space of $n=10$ slots so we use the **modulus operator**:

`table[313%10] = 'cheese';`  (a%b is the *remainder* of a/b)

Now table looks like [undefined, undefined, undefined, 'cheese',
undefined, undefined, undefined, undefined,
undefined, undefined].

...In general, we set `table[hash(key)%n] = value`.

Application - Hash tables (Learn by example!)

```
Reminder: table = [undefined, undefined, undefined, 'cheese',  
                  undefined, undefined, undefined, undefined,  
                  undefined, undefined].
```

To look up the value for the key 'MOON':

First calculate the hash of the key: $\text{hash}('MOON') = 77+79+79+78 = 313$.

Now return `table[313%10]`. This gets 'cheese' out of `table[3]` where we left it.

...In general, we return `table[hash(key)%n]`.

Application - Hash tables (Learn by example!)

```
Reminder: table = [undefined, undefined, undefined, 'cheese',  
                  undefined, undefined, undefined, undefined,  
                  undefined, undefined].
```

Now let's insert:

Key: 'NEPTUNE', Value: 'seawater'.

First calculate the hash of the key:

$\text{hash}('NEPTUNE') = 78 + 69 + 80 + 84 + 85 + 78 + 69 = 543$.

We evaluate $543 \% 10$ to figure out which index we'll store 'seawater' in.

Hold up! $543 \% 10$ is 3, and we **already have something in table[3]**. This is a **hash collision**. Maybe things aren't quite as simple as we thought...

Application - Hash tables redux

Our approach up to this point only works if every key we insert hashes to a different location in the table. This is up to pure chance and will *almost certainly* not remain true for long as the table starts to fill up.

Here are a few common approaches for handling hash collisions:

- **Linear probing**
- **Double hashing**
- **Chaining**

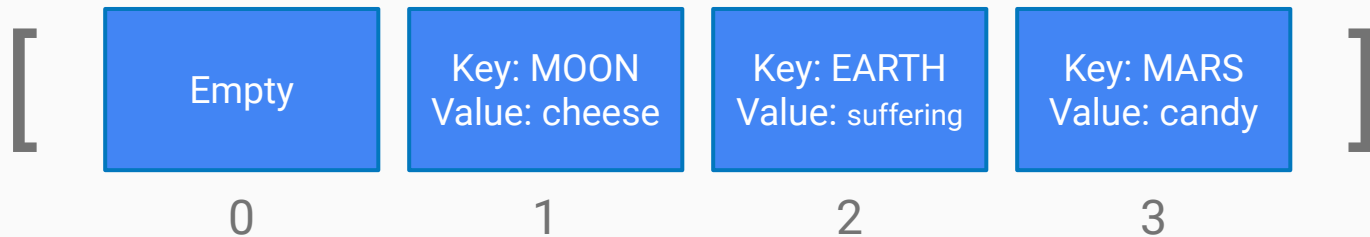
Application - Hash tables (linear probing)

In **linear probing**, when we insert into the table at a spot that's already full, we just try the next spot... and the next... and the next... until we find a free space.

Similarly we when we look up a key, we have to check the spot determined by the hash function... and the next... and the next... until we find the key we're looking for (or an empty space).

NB: Now we have to store the values *and* the keys in the table so that we can identify the correct value when we want to look them up later!)

Application - Hash tables (linear probing)



Say we're inserting `<VENUS, shell>` and `hash('VENUS')%4 = 1`.

We would like to insert at `table[1]` but it's full...

...so we try `table[2]` but it's full...

...so we try `table[3]` but it's full...

...so we wrap around and try `table[0]` and we find a space!

In the end, we put `<VENUS, shell>` into `table[0]`.

Application - Hash tables (linear probing)

[<div>Key: VENUS Value: shell</div>	<div>Key: MOON Value: cheese</div>	<div>Key: EARTH Value: suffering</div>	<div>Key: MARS Value: candy</div>]
	0	1	2	3	

We successfully inserted VENUS but we had to probe through the *entire array* to find an open space. 😬

Let's try retrieving the key MERCURY. Say that `hash('MERCURY')%4 = 0`.

We check `table[0]` and it's not MERCURY. But maybe that space was just full when we inserted MERCURY earlier? So we have to check `table[1]` too. Nope. Next we check `table[2]`. And `table[3]`. Finally we've checked them all and can say for sure that MERCURY is not in the table.

In fact, as the “load factor” of the table approaches 1 (all spaces full), *worst-case* insert and retrieve cost approaches $O(n)$. Don't let your table get too full...

Application - Hash tables

When the table fills up (or the load factor crosses some predetermined threshold) you'll need to **resize the table**.

Allocate a new array and reinsert everything again. **Every key should be re-hashed**, to account for the new array size.

Rules of thumb:

- Rehash when the load factor reaches 0.75.
- Double the size of the hash table each time it needs to grow.

Application - Hash tables (double hashing)

Double hashing makes use of two *different* hash functions, hash1 and hash2 .

First we try to insert at $\text{hash1}(\text{key}) \% n$. If that's full, we jump forward by $\text{hash2}(\text{key})$ spaces and try again. We keep jumping forward by increments of $\text{hash2}(\text{key})$ with each attempt.

This is the same as linear probing but with fixed increments of 1 replaced with increments of $\text{hash2}(\text{key})$.

Application - Hash tables (double hashing)

Con: Double hashing breaks **cache locality** by jumping all over the place.

Pro: Double hashing mitigates the **clustering** phenomenon.

Linear probing tends to accumulate performance-killing “clusters” of filled spots:



The longer a “cluster” gets, the likelier it is that a newly-inserted key will hash somewhere inside it. Then the new key gets shunted to the end and the cluster grows even bigger, absorbing other clusters in the way and growing huge! 😬

Double hashing gives us *somewhere else* to go rather than the end of the cluster.

Note that since the “somewhere else” depends on hash2, we’ll go off to somewhere *different* for each key that had collided under the image of hash1.

Application - Hash tables

Linear probing and double hashing are called **open-addressing methods**. They're very efficient— particularly linear probing (provided the load factor is low).

But there's one critical operation we've overlooked so far: *deleting* items from the hash table. This is kind of a headache for open-addressing methods:

What happens if we delete an element from inside a “cluster” of collisions?

Oh. 🤔

Application - Hash tables

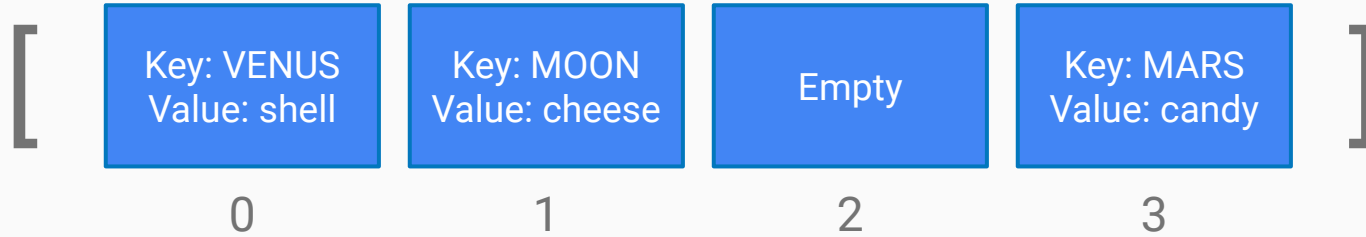
[Key: VENUS Value: shell	Key: MOON Value: cheese	Key: EARTH Value: suffering	Key: MARS Value: candy]
	0	1	2	3	

Remember that in this example $\text{hash}(' \text{VENUS} ') \% 4$ was 1 but we linearly probed to 2, then 3, then 0 to find a space for it. Say that now we delete EARTH.

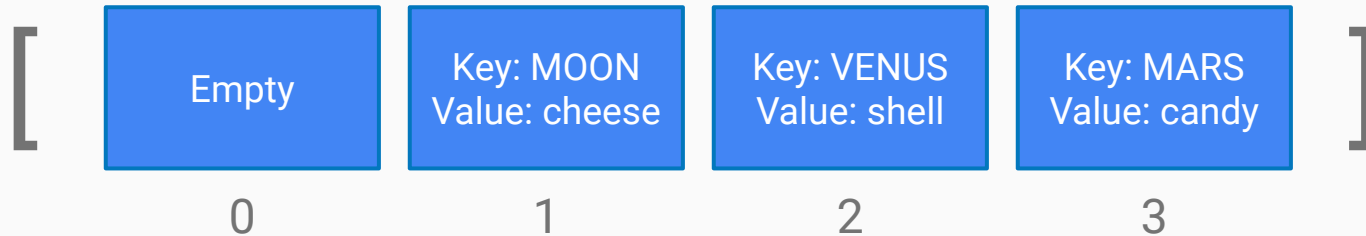
[Key: VENUS Value: shell	Key: MOON Value: cheese	Empty	Key: MARS Value: candy]
	0	1	2	3	

When we retrieve VENUS now, we'll look at `table[1]`, then `table[2]` and say we're done because it's empty. What do we do? We can't keep going; if our strategy were to probe through the entire table every time then we would get worst-case $O(n)$ lookups even with a totally empty hash table! 🙄

Application - Hash tables



The trick is that after deleting EARTH we probe ahead for a key that hashes to something that could *fill in that empty space*. For example, say that $\text{hash}('MARS') \% 4 = 3$ so it's in the correct spot. We have to keep probing. Next up is VENUS, and we know that $\text{hash}('VENUS') \% 4 = 2$ so let's move it:



Now we have a *new* empty space. We keep repeating the process until we reach an empty space.

Application - Hash tables (chaining)

With **chaining**, each “spot” simply holds the head of a *linked list* of all of the `<key, value>` pairs that hashed to that spot.

Pros: Simple to understand and implement! 😊

Cons: Linked lists are slow compared to one big flat table array. 😞

To insert `<key, value>`, check the linked list at `table[hash(key)%n]` to see if key is already there- if so update its value, if not add `<key, value>` to the list.

To retrieve the value for key, look for key in the list at `table[hash(key)%n]`.

Application - Hash tables (iteration)

To provide for . . . of iteration, JavaScript's built-in Map class keeps a separate array to hold all `<key, value>` pairs in the original insertion order.

This adds $O(n)$ overhead to set and remove operations but it's too useful not to have!

Application - Hash tables

What are we hashing anyway? Our hash table keys could be strings (use an off-the-shelf hash function like murmurhash), or numbers (maybe use the numbers themselves as the hash— if they're uniformly distributed!), or *objects*.

If we want to use objects as keys, do we care about:

- **Object identity** (are two objects **the same object**), or
- **Deep equality** (do two objects have **the same content**)?

Application - Hash tables

To key off of an object's *identity*, like Map does, we'd need to key off its memory address, which is never accessible in JavaScript. As a workaround, give each object an auto-incrementing ID at construction time, and key off that instead.

```
1  class MyData {
2    static #OBJECT_ID_COUNT = 0;
3
4    objectID;
5    constructor() {
6      this.objectID = MyData.#OBJECT_ID_COUNT++;
7    }
8  }
9
10 const HashMap = require("../HashMap.js");
11 const map = new HashMap(obj => obj.objectID);
```